Efficient Algorithms for Temporal Balanced Graph Partitioning of Datacenter Workloads

Aleksander Figiel* André Nichterlein* Stefan Schmid*

Abstract. The popularity of distributed machine learning applications and hardware training imposes increasingly stringent performance requirements on the interconnecting communication network. A clever scheduling of the computational workload has the potential to greatly improve datacenter resource utilization, by keeping frequently communicating nodes topologically close. A fundamental underlying optimization problem is known as (static) balanced graph partitioning: How to partition a graph (describing a workload) into equally-sized subgraphs ("clusters") to minimize the number of inter-cluster edges?

We study a temporal version where edges between nodes denote requests, and where clusters can be adjusted over time. Over the last years, several fundamental results have been obtained in the online setting. Motivated by the fact that machine learning workloads are often repetitive and fairly predictable, we focus on the dynamic offline problem variant. In particular, we analyze the border of tractability, obtaining hardness and polynomial-time algorithms for special cases. We further develop effective heuristics based on a novel approximation algorithm that is efficient on static graphs of low treewidth. We complement our theoretical insights with an empirical study of our algorithms on real-world datacenter workloads, and shed light on how much knowledge about future demands is required to improve the performance of online algorithms.

Source code: https://zenodo.org/records/17369617

1 Introduction The remarkable success of artificial intelligence and machine learning applications in a wide variety of fields has led to a significant increase of resource requirements in datacenters. Indeed, over the last years, it became apparent that traditional approaches designed for big data or high-performance computing workloads are insufficient to support machine and deep learning workloads to fully utilize the GPU resources. Accordingly, great efforts are currently being made to improve the performance of GPU datacenters and its communication network connecting distributed compute units and accelerators.

An important approach to improve datacenter network utilization and hence throughput is to tailor schedulers to their specific workloads. In particular, by allocating frequently communicating nodes (processes or GPUs) close to each other (e.g., in the same datacenter rack), communication traffic can be reduced significantly. This allocation may be adapted dynamically, if the workload evolves over time. In such a dynamic scenario, however, we need to trade off the benefits (e.g., reduced communication traffic) and costs (e.g., service interruptions) of adjustments.

A fundamental algorithmic problem underlying such optimizations can be modeled as a dynamic graph partitioning problem, where nodes are dynamically mapped to clusters (e.g., GPUs, servers or racks in a datacenter), aiming to minimize inter-cluster communication while accounting for migration costs, given a time-evolving demand. Over the last years, several fundamental results have been obtained for this problem, however, most previous work focuses on an online setting where the demand is not known ahead of time and where the online algorithm can profit from resource augmentation (unlike the optimal offline algorithm it is compared to).

In this paper we are particularly interested in the offline version of the problem, where the demand changes over time but is known ahead of time. This is an important model in practice, as especially machine learning workloads are often repetitive and hence predictable, featuring temporal structure [4, 29].

Model. A set V of vertices represents, for example, processes which communicate with each other over time. Discretizing time into $\tau \geq 1$ time steps, we use τ edge sets $E_1, \ldots, E_\tau$ over V, where an edge $e = \{u, v\} \in E_t$ represents communication between the vertices u and v at time step t. This results in a temporal graph $G = (V, E_1, \ldots, E_\tau)$ with $n = |V|$ vertices. If $\tau = 1$, then the temporal graph is a classic graph (all our graphs are undirected and simple).

The objective is to partition G for each time step into ℓ clusters with at most k vertices each. We penalize every edge having endpoints in different clusters (modeling higher communication cost) and every vertex mov-

*Technische Universität Berlin, Germany

ing to a different cluster (modeling migration cost). Formally, we consider the problem Temporal (Almost) Balanced Graph Partitioning (TABGP) (where we use $[h] := \{1, \dots, h\}$ for $h \in \mathbb{N}$).

TABGP

Input: A temporal graph $G = (V, E_1, \dots, E_\tau)$ and numbers $k, \ell \in \mathbb{N}$, $\alpha \in \mathbb{Q}^+$

Question: Find τ partitions $V_t = V_t^1 \cup \dots V_t^\ell$ for $t \in [\tau]$ with $|V_t^i| \leq k$ that minimize

$$s = \sum_{t \in [\tau]} |\{\{u, v\} \in E_t \mid u \in V_t^i, v \in V_t^j, i \neq j\}| + \alpha \sum_{t \in [\tau-1]} \sum_{i \in [\ell]} |V_t^i \setminus V_{t+1}^i|.$$

The above formalization ensures that moving a vertex from one cluster V_t^i to another cluster V_t^j in the next time step will incur a cost of α. The special case $\tau = 1$ is called (Almost) Balanced Graph Partitioning (ABGP). Note that $k \geq \lceil n/\ell \rceil$ and the larger k is the more slack one has in solving the problem (hence the "almost" in the problem name).

Our Contributions. We advance the theoretical and practical understanding of TABGP. In particular, we study its computational complexity on restricted graphs. We obtain polynomial-time algorithms on paths for a bounded number of time steps or a bounded number of connected clusters. Complementing these results, we prove NP-hardness on partial grids already for ABGP (i.e., $\tau = 1$). Due to this intractability, we turn to approximation. We provide a polynomial-time algorithm for ABGP that computes a partitioning of optimal cost but the clusters may be of twice the optimum size. Starting from this approximation, we use established local-search techniques to tackle the temporal problem variant.

We further demonstrate the effectiveness of our algorithms in simulations based on real-world datacenter workloads. Our simulations also shed light on the question of how much knowledge about future demand can help improve the performance of online algorithms. Moreover, with a simple parameter setting, our algorithm can be adapted to be an online algorithm (without theoretical performance guarantees) and compares favorable to existing online algorithms that do have a bounded competitive ratio.

Related Work. For static scenarios in which the workload does not evolve (i. e. $\tau = 1$), Balanced Graph Partitioning (also studied under the name $\mathcal{T}_{k+1}$-Free Edge Deletion) has been studied for a long time. The book by Garey and Johnson [14] already covers the NP-hardness of Minimum Bisection (partition a graph into two equal size clusters minimizing the number of edges between these clusters). For a non-constant number of clusters the problem is known to be NP-hard even on trees [12]. In contrast, for constant size clusters and constant-treewidth graphs the problem is linear-time solvable [10].

Our temporal model has also been studied by Räcke et al. [23, 24, 25] who consider it as an offline version of a dynamic graph partitioning problem (which we discuss below). Räcke et al. [23] presented a polynomial-time $O(\log n)$-approximation algorithm that uses resource augmentation (i. e. their approximation algorithm can put more vertices in a cluster than an optimum solution is allowed to), using LP relaxation and Bartal's clustering algorithm to round it. However, these results are purely of theoretical interest so far: We tested their LP. It did not finish within hours even on ten-vertex graphs and the clustering algorithm is rather sophisticated as well.

The dynamic balanced graph partitioning problem was introduced by Avin et al. [1, 5]. The authors considered an online setting and showed that no deterministic online algorithm can achieve a competitive ratio better than $\Omega(n)$, even in a model with constant-factor, $(1 + \varepsilon)$ augmentation (as long as $\varepsilon < 1$ [2]) and if the demand graph forms a ring. Avin et al. also presented a deterministic online algorithm which achieves a competitive ratio of $O(k \log k)$ with a constant factor augmentation, and Rajaraman and Wasim showed an $O(n \log n)$-competitive deterministic algorithm for a fixed $\varepsilon > 0$ [26]. Avin et al.'s algorithm however relies on expensive repartitioning operations and has a super-polynomial runtime. Forner et al. [13] later showed that a competitive ratio of $O(k \log k)$ can also be achieved with a polynomial-time online algorithm which monitors the connectivity of communication requests over time, rather than the density. Moreover, they provide an implementation of their algorithm that we also compare against in our computational experiments. Pacut et al. [21] contributed an $O(\ell)$-competitive online algorithm for a scenario without resource augmentation and the case where $k = 3$. Deterministic online algorithms for the ring communication pattern have also been studied in a model where the adversary needs to generate the communication sequence from a random distribution in an *i.i.d.* manner [3]: in this scenario, it has been shown that even deterministic algorithms can achieve a polylogarithmic competitive ratio. The online version of the problem is however different from our setting and the corresponding algorithms and techniques are not applicable.

So far, to the best of our knowledge, randomized online algorithms have only been studied in the learning

variant introduced by Henzinger et al. [16, 17]. In their first paper on the learning variant, Henzinger et al. [17] still only studied deterministic algorithms and presented a deterministic exponential-time algorithm with competitive ratio $O(\ell \log \ell \log k)$ as well as a lower bound of $\Omega(\log k)$ on the competitive ratio of any deterministic online algorithm. While their derived bounds are tight for $\ell = O(1)$ servers, there remains a gap of factor $O(\ell \log \ell)$ between upper and lower bound for the scenario of $\ell = \omega(1)$. Henzinger et al. [16] presented deterministic and randomized algorithms which achieve (almost) tight bounds for the learning variant. In particular, a polynomial-time randomized algorithm is described which achieves a polylogarithmic competitive ratio of $O(\log \ell + \log k)$; it is proved that no randomized online algorithm can achieve a lower competitive ratio. Their approach establishes and exploits a connection to generalized online scheduling, in particular, the works by Hochbaum and Shmoys [18] and Sanders et al. [27].

More generally, our model is related to dynamic bin packing problems which allow for limited *repacking* [11]: this model can be seen as a variant of our problem where pieces (resp. items) can both be dynamically inserted and deleted, and it is also possible to open new servers (i.e., bins); the goal is to use only an (almost) minimal number of servers, and to minimize the number of piece (resp. item) moves. However, the techniques of Feldkord et al. [11] do not extend to our problem.

In a model without augmentation and for general demands, until recently, the best known online algorithm had a competitive ratio of $O(n^2)$ [5]. Perhaps surprisingly, not even a randomized algorithm was known which achieves a better competitive ratio. Bienkowski and Schmid [7] very recently showed that in fact, (slightly) subquadratic competitive ratios are possible in this model. However, there remains a significant gap between upper and lower bound.

Organization. We analyze the computational complexity and polynomial-time solvability of special cases of ABGP and TABGP in Section 2. Our main algorithm is presented in section 3 and empirically evaluated in section 4.

2 Computational Complexity of TABGP

In this section, we discuss the computational complexity of TABGP and identify tractable special cases. For observing that TABGP is NP-hard, we can restrict ourselves to the non-temporal variant: Indeed, ABGP is NP-hard, even if the input graph is a tree and has either constant maximum degree or constant diameter [12]. Reducing from 3-Partition or Bin Packing (both are strongly NP-hard [14]) where a number h is encoded as path on h vertices yields the following result:

Observation 2.1. (Almost) Balanced Graph Partitioning *is NP-hard even if* $s = 0$ *and* $G = (V, E)$ *is a collection of paths.*

Faced with this intractability, we tweak the problem in the hopes of obtaining algorithms. It is conceivable that the clusters are well-connected in real-world instances. Thus, we require the clusters to be connected. Note that this circumvents the above NP-hardness as the clusters are not connected in the respective constructions. We start with simple graph classes that display regular communication patterns like paths or partial grids to explore islands of tractability.

Paths. We first observe that in contrast to Observation 2.1, ABGP is trivial if the input is a single path (cut into subpaths on k vertices). The problem remains tractable also in the temporal setting, at least to some extent: we show that, assuming our partitions "respect" the path structure, TABGP is polynomial-time solvable if the number of clusters or the number of timesteps is constant. Due to space limitations, we defer the proof to a full version.

Theorem 2.2. *If* $G^{\cup} := (V, \bigcup_{i \in [\tau]} E_i)$ *is a path, each cluster is a connected component in* $G^{\cup}$*, and either* τ *or* ℓ *is constant, then* TABGP *is polynomial time solvable.*

Partial Grids. The restriction to paths yielded some algorithmic results. In contrast, we next show that the problem is already hard on partial grids. A partial grid is a subgraph of a grid, that is, the vertices can be represented by points with integer coordinates and two vertices can be adjacent if they agree on one coordinate and differ by one in the other coordinate.

Theorem 2.3. ABGP *remains NP-hard on partial grids even if all clusters need to be connected.*

Proof. We reduce from Vertex Cover. Let (G, h) be the instance of Vertex Cover with G being a planar graph of maximum degree three; note that Vertex Cover remains NP-complete on these graphs [14]. We construct our instance (G', ℓ, k, b) of (the decision variant of) ABGP (where b denotes the bound on the cost and the factor α is obsolete) as follows: We "embed" G onto a grid G' so that the edges of G are represented by paths in G', i. e., we use a rectilinear embedding which can be computed in polynomial time [28]. We then replace each vertex v by a gadget that consists of two subgrids on the vertices A_v, B_v connected via a single edge, see Figure 2.1 for an illustration. The sizes are chosen such that $|A_v| + |B_v| = k$ and $|A_v| > |B_v|$. For all vertices in V, the corresponding subgrids will have the same size. The edges incident to v in G dock onto A_v. We also add small subgrids to the edges to

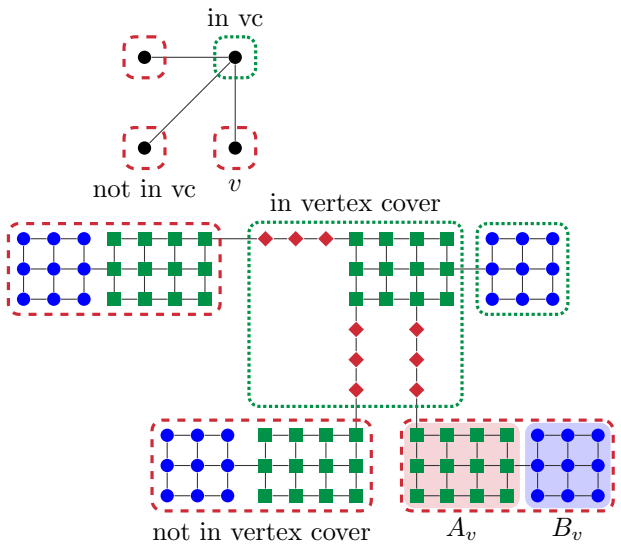

Figure 2.1: Example of our construction showing NP-hardness on partial grids. On the top is a graph on four vertices denoting the Vertex Cover instance (with an optimal vertex cover indicated by the green dotted line). On the bottom is the constructed instance of ABGP where the partition is indicated by the boxes (dashed and dotted to indicate the connection to the vertex cover). The different clusters of the construction (two subgrids A_v, B_v, and edge vertices) are indicated by the different node shapes. For the (bottom right) vertex v in the top graph the two clusters A_v and B_v are highlighted in the bottom graph.

ensure that they all have the same size c where c is chosen so that $|A_v| + 3c \leq k$. Selecting k big enough e.g. $k = n^4$ ensures that $|A_v|$, $|B_v|$, and c can be chosen to satisfy the above constraints. Finally, set $b := m + h$ and $\ell = n + h$.

The correctness of the construction can be seen as follows.

If there is a vertex cover S of size h in G, then delete the following edges in G': For each vertex v in S, remove the edge connecting A_v and B_v. For each edge e in G remove the connection to subgrid A_v where v is not contained in S. If both endpoints are in S, remove the connection to either one of the two subgrids. Thus, we remove b edges. Moreover, since $|A_v| + 3c \leq k$, each connected of the ℓ components contains at most k vertices.

Reversely, let F' be a set of at most b edges, such that $G - F'$ has at most ℓ connected components of at most k vertices each. Observe that no two A-subgrids A_u and A_v can end up in the same connected component as this results in more than k vertices. Thus, for every edge e in G the set F' has to contain at least one edge corresponding to e. Hence, at most $b - m = h$ edges in F' can be used to remove A-subgrids from B-subgrids. Furthermore, if for a vertex v a vertex gadget remains unchanged, i. e., A_v and B_v are not disconnected, then all edges incident to v in G need to be disconnected in G' from A_v. This implies, that the at most h vertex gadgets where A_v is disconnected from B_v correspond to a vertex cover of at most h vertices in G. □

3 Algorithm for TABGP Given our algorithmic results on exact solutions for special problem variants as well as our hardness results, we next consider how to solve the ABGP problem approximately. In particular, we will propose an approximation algorithm and a heuristic in the following. We start with the non-temporal variant (one time step) on graphs of constant treewidth and design some approximation algorithm (recall that the problem is NP-hard on trees). Afterwards, we extend the algorithm to a heuristic for the temporal variant.

2-Phase approach for ABGP. Faced with the computational hardness of (Almost) Balanced Graph Partitioning on trees [12] and on grids (see Theorem 2.3), we turn to approximation algorithms where we approximate not the cost (number of cut edges) but the size of the partitions. More precisely, we will provide an algorithm that produces a partition that cuts at most as many edges as an optimal solution but creates clusters that can be larger by a factor of at most 2; this is called resource augmentation.

To this end, we will employ a two-phase algorithm.

1. Remove as few edges as possible to get connected components of size at most k vertices each. This problem is also known as $\mathcal{T}_{k+1}$-Free Edge Deletion where $\mathcal{T}_{k+1}$ denotes the set of all trees on $k+1$ vertices. $\mathcal{T}_{k+1}$-Free Edge Deletion is NP-hard and can be solved in $O((\omega k)^{2\omega} n)$ time on graphs with treewidth at most ω [10]. (Enright and Meeks [10] conjecture the problem to be W[1]-hard with respect to treewidth alone, which was recently confirmed [6]. Hence the dependency on k in this running time is presumably unavoidable.)

   T_{k+1}-free Deletion
   Input: A graph $G = (V, E)$ and a non-negative integer h.
   Question: Can h edges be deleted such that each connected component has size at most k?

2. Pack the connected components to the servers. This problem corresponds to the scheduling prob-

lem $P||C_{\max}$, i. e. the classical problem of makespan minimization for identical parallel machines in scheduling, defined as follows:

$P||C_{\max}$
Input: A set $\mathcal{M}$ machines and a set $\mathcal{J}$ jobs, each job j has a processing time p_j.
Question: Find the assignment of jobs to machines minimizing the maximum load $C_{\max}$ with:

$$C_{\max} := \min_{A:\mathcal{J}\to\mathcal{M}} \max_{m\in\mathcal{M}} \sum_{j\in\mathcal{J}:A(j)=m} p_j.$$

In our setting each connected component is a job with the number of vertices denoting the processing time.

Using textbook approximation algorithms and their analysis for $P||C_{\max}$, we can derive bounds on the cluster size for the above algorithm.

LEMMA 3.1. *Given a graph where each connected component has order at most k, one can compute in polynomial time a packing of the components on the ℓ servers such that each server contains at most $2k$ vertices.*

Proof. The algorithm is as follows (essentially the standard greedy algorithm for $P||C_{\max}$): Place each unpacked component C into the first server so that the load is at most $2k$.

This algorithm clearly runs in polynomial time and each server stores components with overall at most $2d$ vertices. It remains to show that all components can be packed. To this end, consider an arbitrary component C considered by the algorithm: Observe that at least one server contains $\lfloor \frac{n}{\ell} \rfloor \leq k$ vertices or less as the components comprise of in total n vertices. As each component at most contains k vertices, C can be packed on this server. ∎

We can slightly improve on the size-bound given in the previous lemma by first sorting the clusters by size. In contrast to $P||C_{\max}$ we only get a minor improvement in our case (as our lowerbound is weaker than the lowerbound that can be used in $P||C_{\max}$). Formally, we obtain:

LEMMA 3.2. *Given a graph where each connected component has order at most $k \geq \lceil n/\ell \rceil$, one can compute in polynomial time a packing of the components on the ℓ servers such that each server contains at most $\max\{k, (2-2/(\ell+1))n/\ell\}$ vertices.*

Proof. The algorithm is as follows (essentially the standard greedy algorithm for $P||C_{\max}$): First sort the components by number of contained vertices in descending order. Then, following this order, place the components one by one to the server with minimum load so far.

Consider the component C that was added last on the server S storing most vertices at the end of the algorithm. Let $h \leq k$ be the number of vertices in component C. If S did not contain any component yet, then each server contains at most k vertices. Assume S did contain some component(s). Hence, at least ℓ components are distributed before S and $h \leq \lceil n/(\ell+1) \rceil$. When adding C to the server S, it had the least load of all, thus, load at most $\lfloor (n-h)/\ell \rfloor \leq k$. Hence, S contains at most $\lfloor (n-h)/\ell \rfloor + h$ vertices at the end. The load of S is thus

$$\left\lfloor \frac{n-h}{\ell} \right\rfloor + h \leq \frac{n-h}{\ell} + h = \frac{n}{\ell} + \frac{(\ell-1)h}{\ell}$$
$$\leq \frac{n}{\ell} + \frac{(\ell-1)n}{\ell(\ell+1)} = \frac{n}{\ell}\left(2 - \frac{2}{\ell+1}\right). \quad ∎$$

Combining the above lemma with the result of Enright and Meeks [10], we obtain the following approximation result.

THEOREM 3.3. *In graphs of constant treewidth we can compute in polynomial time a solution of optimal cost for* (ALMOST) BALANCED GRAPH PARTITIONING *where each server contains at most $2k - 2k/(\ell+1)$ vertices.*

Proof. The approach is as discussed above. For constant treewidth, the algorithm of Enright and Meeks [10] runs in time $O((\omega k)^{2\omega} n)$ and computes a minimum-cardinality edge set F such that each connected component in $G - F$ contains at most k vertices. Next, we apply Lemma 3.2 to pack these components on the ℓ servers. As $k \geq \lceil n/\ell \rceil$, each server will end up with components totaling at most $2k - 2k/(\ell+1)$ vertices. Note that an optimum solution to (ALMOST) BALANCED GRAPH PARTITIONING will cut at least $|F|$ edges as it can pack at most k vertices on each server. Thus, our solution cuts at most as many edges as an optimal one. ∎

Remark 3.4. Observe that for constant k and constant treewidth, our approximation algorithm runs in quasi-linear time: For these values the algorithm of Enright and Meeks [10] runs in linear time and the most expensive step in the algorithm behind Lemma 3.2 is the sorting and maintaining the priority queue. Both are doable in $O(n \log n)$ time overall.

Extension to temporal variant. To extend the above algorithm to work for more than one time step, we employ a local search approach: we maintain a partition of the vertex set and look for improving this partition by moving vertices to different clusters. While this local search approach achieves exceptional results in practice for partitioning (i. e. clustering) problems [8, 19], it is hard to theoretically analyze [9].

The input to our local search is is an edge-weighted temporal graph and a clustering for each time step. In one local search step we select a single timestep and try to move a single vertex into a different cluster that has free capacity or swap two vertices from different clusters. The swaps and moves are only done if they decreases the cost of the clustering including the migration costs. This is repeated until the clustering does not change. The edge-weights allow us to employ local search in different variants as described below:

1. We compress the first x timesteps into one as follows: each edge gets a weight equal to how often the edge appears in the first timesteps. On this weighted graph a static clustering is computed. For the remaining timesteps the clustering of the preceding timestep is taken and improved by local search. Inhere, the local search takes the compressed x previous and next timesteps to have some foresight and limit too frequent migration of vertices between clusters. This variant is denoted by Roll_x. We use $x \in \{0, 1, 10\}$. The variant $x = 0$ does not use any knowledge of the future and can be seen as online heuristic; we thus call this Online.

2. We compress all timesteps into one, i. e., the edge weights denote the number of occurrences of this edge in the temporal graph. On this weighted static graph a clustering is computed. Then we continue iteratively as follows: We split one compressed timestep into two, each new timestep compresses one half of the original timesteps. The clustering for the old compressed timestep is taken as initial solution for the two new timesteps. This initial solution is improved by local search. We repeat this process until a compressed timestep encompasses only one timestep. This variant is denoted by Refine.

For all variants except Online there is a final local search step in the end where each solution in all timesteps combined is optimized.

We are unable to provide theoretical guarantees for our heuristic for the temporal setting, and thus we focus on experimental evaluation. As the non-temporal variant is computationally quite challenging (see section 2), it is conceivable that the harder temporal problem variant does not allow for results similar to Theorem 3.3.

Algorithm 3.1 Heuristic for finding dense clusters

procedure DENSE(static graph G, ℓ, k)
  $C = [\,]$ ▷ empty list
  for ℓ rounds **do**
    $X = \emptyset$
    Define **score**$(v) = (|N(v) \cap X|, -\deg(v))$
    while $|X| < k$ and unused vertices exist **do**
      $v \leftarrow$ unused vertex with largest **score**(v)
      add v to X and mark v as used
        ▷ scores are compared lexicographically
    end while
    append X to C
  end for
  return C
  ▷ After adding a vertex v to X only the scores of neighbors of v change. Utilizing heap based priority queues one can obtain $O(n + m \log n)$ running time.
end procedure

Implementation Details. While we generally follow our 2-step-approach to compute good solutions for (ALMOST) BALANCED GRAPH PARTITIONING, the dynamic programming behind step one with a running time of $O((\omega k)^{2\omega} n)$ is too time (and memory) consuming for our real-world graphs. Thus, we replace the dynamic program by the following heuristic: We greedily compute a densest subgraph (a subgraph maximizing the ratio of edges and vertices) of k vertices and add it as cluster to our solution. This is iteratively repeated until the remaining graph has at most k vertices, in which case it forms the last cluster. Note that computing a densest subgraph is polynomial-time solvable with flow-techniques [15, 22]. However, computing a densest subgraph of a given size is NP-hard, as it contains CLIQUE as special case. We call this heuristic Dense. The pseudocode is given in Algorithm 3.1.

As a baseline to compare our clustering heuristic for this static case, we use and compare to the commonly used METIS-library. The corresponding heuristics (METIS with or without our local search extensions) start with METIS. We utilize the default parameters of METIS.

4 Empirical Insights In order to study the performance of our heuristics under realistic datacenter workloads we conducted simulations with traffic traces from Meta (formerly Facebook) and HPC as well as LLM applications. We compare our heuristics to an optimal (super-polynomial time) exact solution, as well as to the state-of-the-art online implementation dCrep

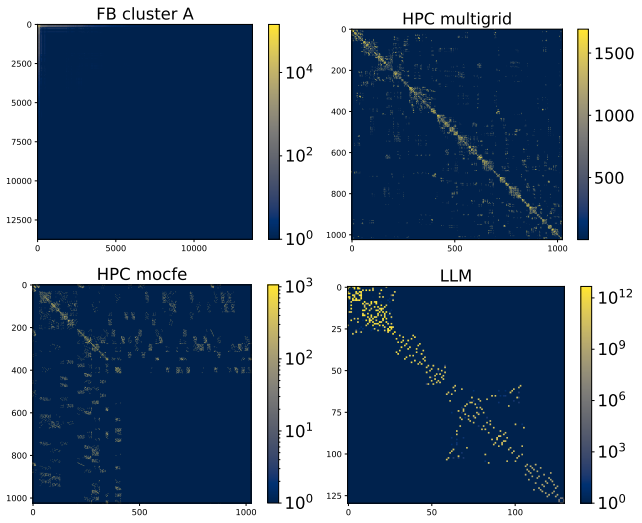

Figure 4.1: Heatmaps of the edge weights in some real-world traces. The Facebook cluster A (top left) has a very few high-degree vertices shown in the top left of the heatmap. The LLM trace (bottom right) by far the highest edge weights and its structure is visibly distinct.

of Forner et al. [13].

4.1 Workloads. We use a machine learning workload from the Zhejiang Lab LLM Trace Program where a trace for one day is publicly available [20]. Additionally, we consider two realistic workloads, one from Facebook and one from a high-performance computing (HPC) applications with overall 7 traces; see Figure 4.1 for heatmaps of the edge-weight distributions. Due to space restrictions, we highlight exemplary results on the Facebook cluster A (rack) and HPC mocfe traces. The latter two workloads are publicly available and have been used in the literature before [4]. We discretize the data set by creating temporal graphs with between 10 and 100 time steps. Each discretization partitions the timescale in the respective number of equal-length time intervals and assigns the edges to the time steps where the edge occurred at least once in the respective interval.

Finally, we also consider a synthetic workload, which allows us to study the performance of our heuristics under different demand densities. To this end, we use Erdős-Rényi random graphs. We use this workload to compare our performance against an exact solution; since the latter cannot be computed in polynomial time, we study small graphs only: The random graphs have 12 vertices and edge probability varying between 0.1 and 0.9 (in 0.1 steps). Per instance, we created 3 graphs with each of them representing one time step. For each density, we create 50 graphs and report average values. Slightly larger instances already proved too challenging to solve exactly, with no solution after 2 weeks of computation.

4.2 ILP. We developed the following ILP formulation to compute the optimum solutions. The ILP employs indicator variables $x_{v,i,t}$ that encode if vertex v is on server i in time step t. To encode the cost function additional variables are used for each edge and each vertex at each timestep. Thus, it uses $O(n \cdot \ell \cdot \tau + m \cdot \tau)$ variables. The constraints are as follows:

$$\text{minimize:} \quad \sum_{t\in[\tau]} \sum_{\{u,v\}\in E_t} y_{u,v,t} + \alpha \sum_{t\in[\tau-1]} \sum_{v\in V} z_{v,t}$$

$$\begin{aligned}
\text{such that:} \quad x_{v,i,t} &\in \{0,1\} && \forall v \in V, i \in [\ell], t \in [\tau] \\
y_{u,v,t} &\in \{0,1\} && \forall \{u,v\} \in E_t, t \in [\tau] \\
z_{v,t} &\in \{0,1\} && \forall v \in V_t, t \in [\tau-1] \\
\sum_{i\in[\ell]} x_{v,i,t} &= 1 && \forall v \in V, t \in [\tau] \\
\sum_{v\in V} x_{v,i,t} &\le k && \forall i \in [\ell], t \in [\tau] \\
x_{v,i,t} - x_{u,i,t} &\le y_{u,v,t} && \\
x_{u,i,t} - x_{v,i,t} &\le y_{u,v,t} && \forall \{u,v\} \in E_t, t \in [\tau], i \in [\ell] \\
x_{v,i,t} - x_{v,i,t+1} &\le z_{v,t} && \forall v \in V, t \in [\tau-1], i \in [\ell]
\end{aligned}$$

4.3 Implementation. Our algorithms as well as a benchmark framework were initially written entirely in Python. We identified the more computationally intensive code components, our local search, and re-implemented it in C++. The switch from Python to C++ for those components yielded a roughly 20-times speedup.

Our algorithm implementations also heavily utilize priority queues. In particular, an operation we frequently perform is updating the priority of an enqueued element. Standard libraries in Python or C++ do not provide a direct means of doing so. We wrote a custom heap-based priority queue which performs a priority update in-place and performs sift down and sift up operations to restore the heap property. This takes logarithmic time in the worst case, but can perform much faster when the priority value only changes by a small value, in which case sometimes only few operations are needed to restore the heap property.

4.4 Setup. All experiments were performed on a machine with an AMD EPYC™ 7543 CPU with 1TB of RAM running Ubuntu 24.04 LTS. We used Gurobi

12.0.3 to solve our ILP-formulation. Additionally we provide solutions computed by our heuristics as start solutions for Gurobi.

We set $\alpha = 3$, as was done in previous work by Forner et al. [13]. That is, migrating a vertex to a different cluster costs three times as much as a communication between vertices in different clusters. For real workloads, we set $k = \lceil 2.1 \cdot n/\ell \rceil$ as was used by Forner et al. [13] where 2.1 is the augmentation factor that they use. Their algorithms require an augmentation factor strictly larger than 2. Our heuristics can also be run in the more challenging setting without augmentation, that is, augmentation factor 1 and $k = \lceil n/\ell \rceil$ (which we use in the synthetic workload).

4.5 Experimental results. Our analysis of the experimental results is split into three parts:

1. Comparison against optimum solutions computed with the ILP.
2. Comparison against dCrep from Forner et al. [13] on the three real-world workloads. As dCrep is designed for unweighted graphs, we ignore any edge weights.
3. Detailed analysis for the machine learning trace including edge-weights modeling the communication volumes.

ILP. We first investigate the relative error of our heuristics. Due to the running time limitations of the ILP solver, this only includes graphs with 12 vertices, and at most three time steps. With rising density the relative error of the heuristics generally shrinks, see top plot in Figure 4.2. This is somewhat expected as more edges result in higher cost of the optimum solution. One can also observe that methods that have a better view of the "future" and construct a partitioning at time t while considering more of the subsequent time steps also achieve lower costs. Even though a local search is performed at the end to optimize the partitions (which takes into account all time steps at once, except for Dense-Online), the initial solution before starting local search also plays a big role. Here the Dense-Refine heuristic achieves the lowest costs, and we observe a maximum relative error of 2.9 % and an average relative error of 0.8 %.

We further investigate the impact of the starting heuristic for the static case ($\tau = 1$) as well as the impact of local search on the objective costs, see bottom plot in Figure 4.2. In the static case it can be observed that a naive random partitioning performs only slightly worse than our greedy heuristic when applying local search in both cases. The previously observed power of local search for clustering problems [8, 19] is confirmed in our setting: Nearly optimal solutions are obtained from a random solution.

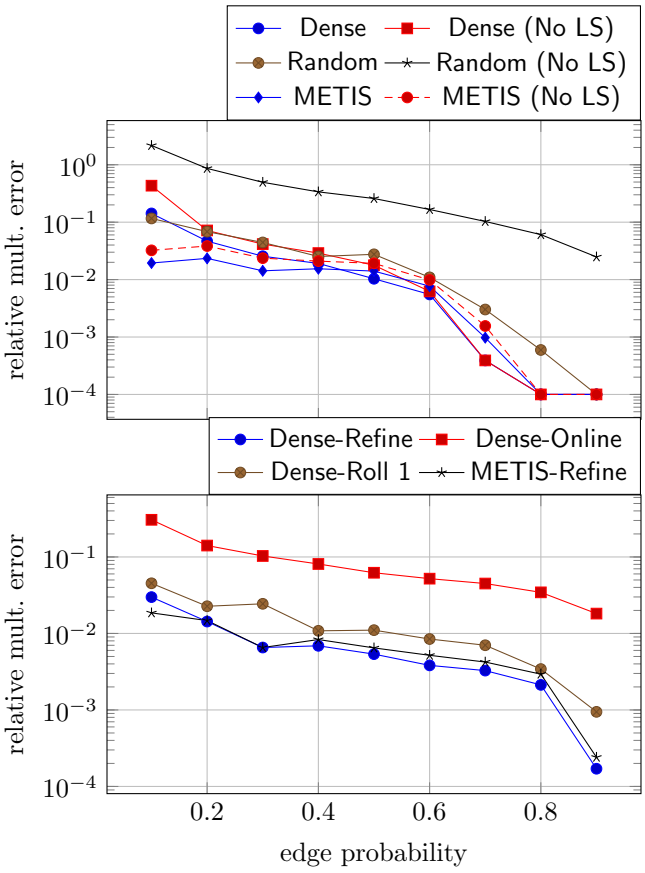

Figure 4.2: Relative error of various heuristics ($+10^{-4}$ to represent 0) on Erdős-Rényi random graphs ($n = 12, \ell = 4, k = \lceil n/\ell \rceil$). The top plot shows the static case $\tau = 1$. The bottom shows the temporal setting with $\tau = 3$.

Note that the LP-relaxation of our ILP-formulation has an optimum value of 0: setting $x_{v,i,t} = 1/\ell$ for all x-variables (that should encode the assignment of vertices to clusters) allows to set all other variables to 0. The LP-relaxation by Räcke et al. [23] is used in their $O(\log n)$-approximation (with augmentation factor 2) and provides much better lower bounds. However, while still having a polynomial number of constraints, this number is prohibitively large: On small random graphs with 10 vertices and 10 time steps, their LP-relaxation did not finish the computation within 10 hours (this is even after adapting the formulation to allow multiple edges per time step to reduce the model size).

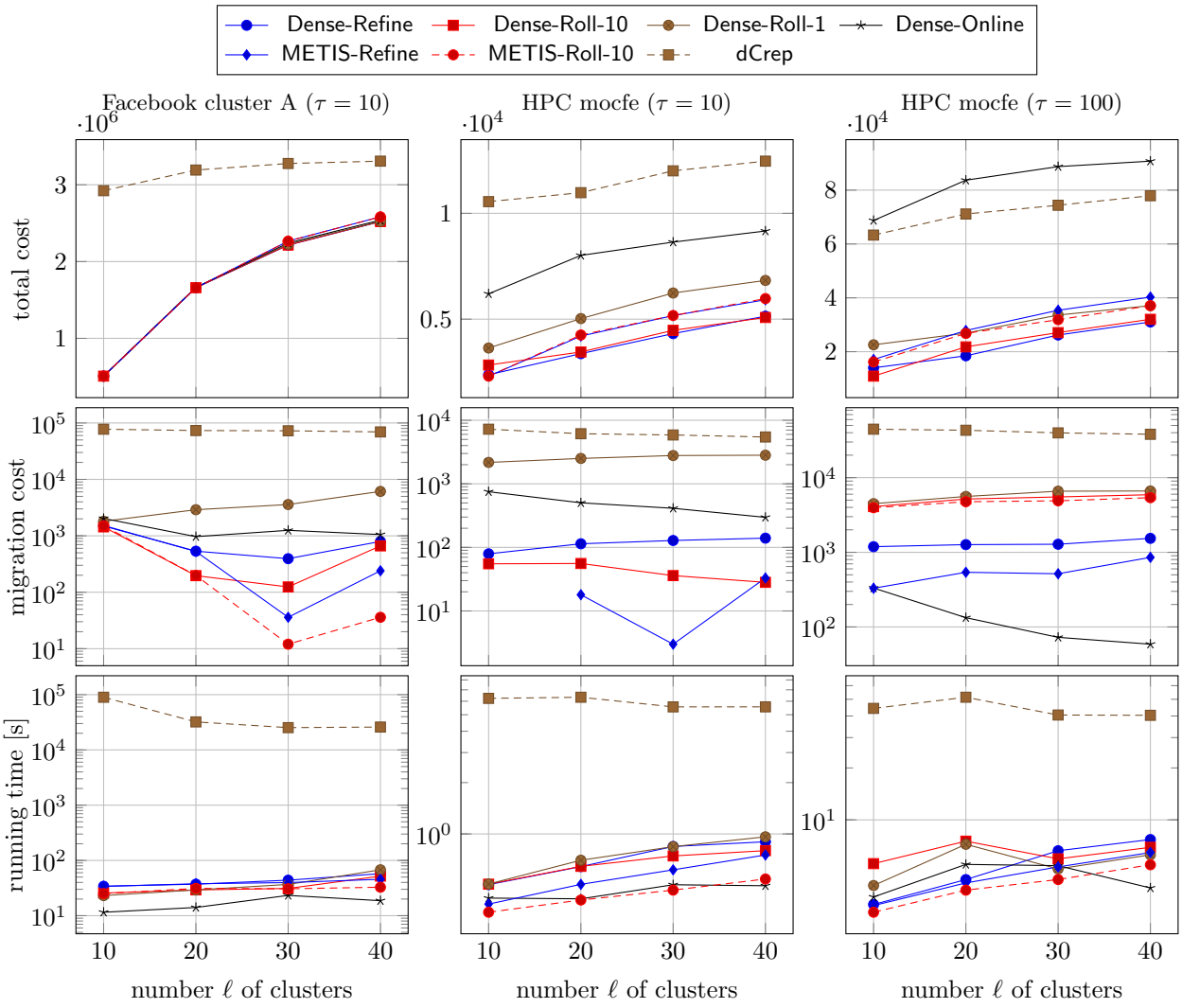

Figure 4.3: Total cost and migration cost of our heuristics on two real-world graphs are displayed in the first two rows of plot. The total cost consists of the communication cost (communication between nodes in different clusters) and migration costs. The plots at the bottom row show the running time. For the Facebook plots we only consider $\tau = 10$ timesteps so that dCrep finishes in reasonable time.

dCrep. On the real-world workloads, we compare our heuristics against each other and against the online algorithm dCrep of Forner et al. [13]. This is by no means a fair comparison: our heuristics have access to the whole instance, the online algorithm does not. dCrep also expects one unweighted edge per timestep. To make the algorithm work in our setting, we give it the edges E_t from time step t one after the other in a random order by starting with $t = 1$. To construct a clustering at time step t we take the clustering of dCrep that is computed after it processed the last edge from E_t. To compensate a bit, we selected the two traces where dCrep performed best for subsequently discussing the results in more detail.

All of our heuristics (except Dense-Online) end up with solutions of roughly the same quality on the real-world workloads. In contrast, dCrep provides solutions that are between 30% and 6 times more costly, see Figure 4.3. Dense-Online also produces on all but one data set (HPC mocfe trace with $\tau = 100$, see top right of Figure 4.3) better solutions than dCrep. Interestingly, the migration costs are often several orders of magnitude smaller than the total cost, which implies that the communication cost clearly dominates the total cost. Although less pronounced, dCrep also admits this pattern. However, our heuristics have a significantly smaller mi-

gration cost than dCrep, see the plots in the second row of Figure 4.3. Thus the question emerges whether, online algorithms could be improved by opting for lowering migration cost?

Finally, we note that the gap between the solutions of dCrep and of our heuristics is smallest on the large dataset from Facebook. However, on this data set the running time of dCrep is prohibitively large (see bottom row of Figure 4.3).

Overall, the running time of dCrep is not much worse than that of our heuristics on the smaller HPC workload (traces having a few hundreds of vertices) with the HPC mocfe being the only trace where for a higher number of clusters the speed of dCrep is actually competitive. In general the running time of dCrep seems to be independent on the number ℓ of clusters, whereas our heuristics get slightly slower with increasing ℓ. In contrast, our heuristics scale significantly better with the size of the traces. On the larger Facebook workload with tens of thousands of vertices the running time of dCrep is measured in *days* while all our heuristics still finish within 25 minutes on even the largest trace. Due to the high running time, we did run dCrep only on the smallest Facebook trace (cluster A), displayed in Figure 4.3 in the left column. Moreover, we estimate that the other pCrep online algorithm by Forner et al. [13] would have taken at least 10 times longer to run.

All our heuristic were initially implemented in Python and not particularly optimized for running time. To demonstrate that our heuristics can be fast enough for practical considerations, we implemented the local search for DENSE-ONLINE in C++ and measured the computation time needed per time step (excluding I/O operations). We selected this heuristic as it is our only one also applicable in the online setting. The results are plotted in Figure 4.4 for all real world traces with the Facebook traces being the largest ones. We observed a roughly 20-times speedup compared to the Python-implementation and reached computation times from mostly one second and less per time step. The initial spikes in the plot come from the Dense heuristic.

Machine Learning Trace. Here we compare our heuristics against each other in more detail on the machine learning trace from Zhejiang Lab that fits best to the problem motivation. As we discard dCrep, we include the integer edge-weights with values exceeding 10^{12} in the trace. Our heuristics can easily deal with edge-weights as we only need to replace counting of edges with summing of edge-weights. For METIS however, these large weights exceed the data type it used for storing edge-weights. We thus create two weighted data sets by either taking the square root or the logarithm

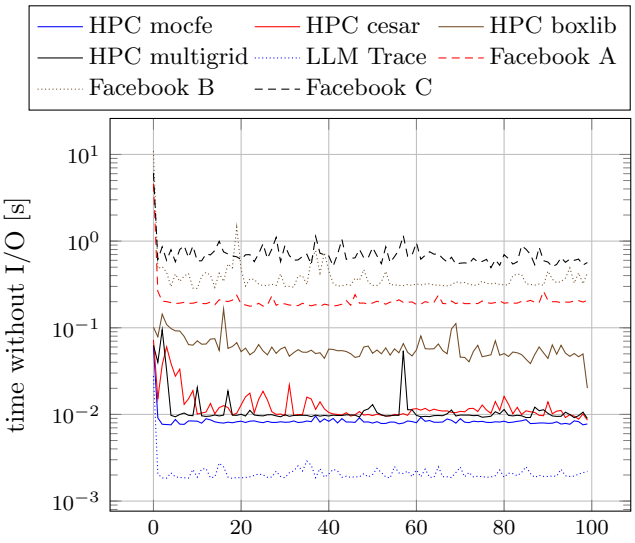

Figure 4.4: Computation time (without the time for I/O's) per time step of Dense-Online on the various real-world traces. The time for computing the initial solution is clearly visible in all plots. The Facebook traces are considerably larger and thus require more time. However, even for the largest trace (Facebook C) the running time stays around roughly one second per time step; which is acceptable.

of the weights as new weight. Moreover, for completeness we also considered the unweighted instance as in previous comparisons. The results are displayed in Figure 4.5. Interestingly, our online heuristic Dense-Online performs comparatively worst in the unweighted setting where the METIS- or Dense-based heuristics perform very similar. In contrast, the higher the edge weights become, the better Dense-Online performs compared to the METIS-based heuristics. We have no explanation for the visible difference of the METIS- and Dense-based heuristics for higher edge-weights.

We remark that the value of $\alpha = 3$ results in the migration costs having a negligible part in the total cost. Our heuristics can deal with different values of α without much change. Moderate changes of α did not change the results significantly. For the LLM trace where we took the square root for all edge-weights, the resulting cost only started to visibly change for $\alpha > 10^3$. In principle, the factor α could depend on the vertices, making the migration cost per vertex part of the input.

5 Future Research While we provided several insights into the tractability and hardness, several interesting areas of the complexity landscape of the balanced graph partitioning problem remain uncharted. In par-

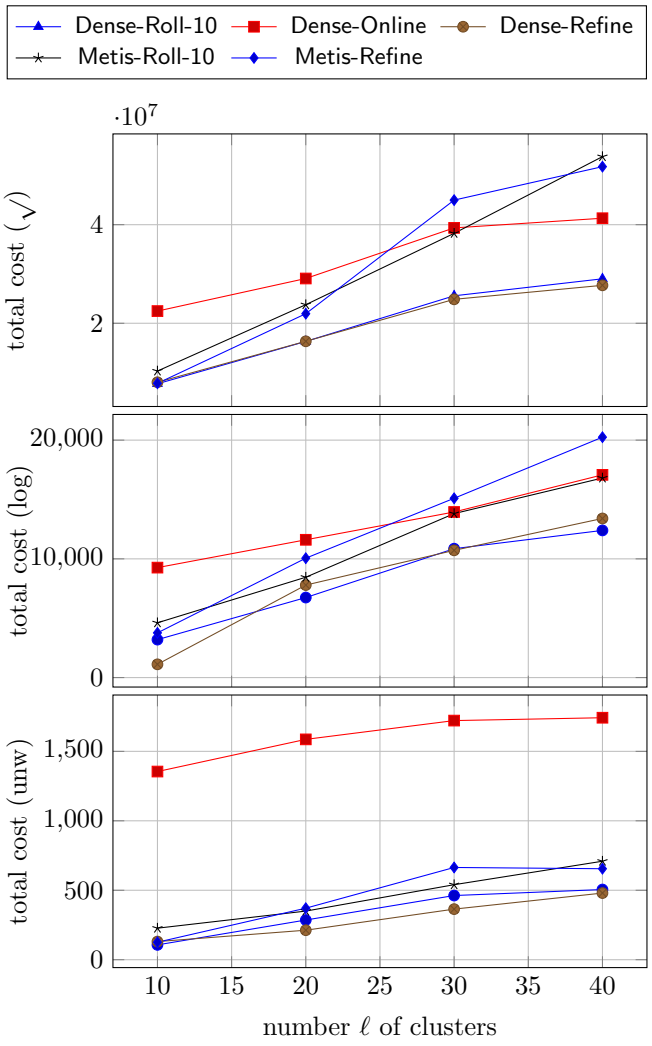

Figure 4.5: Comparison of the heuristics on the machine learning trace with adapted weights and unweighted (bottom plot) respectively.

ticular, it remains to find or extend islands of tractability further, from a parameterized algorithms perspective for exact algorithms, but also for further approximations, or (more likely) combinations thereof.

On the practical side, it will be very interesting to use our algorithms to learn specific collective communication patterns arising in GPU datacenters. As a first step towards achieving improved competitive ratios, can we employ such patterns to obtain $o(n \log n)$ competitive ratios in specific real-world settings?

Acknowledgments Research in part supported by the European Research Council (ERC) Consolidator, grant agreement No. 864228 (AdjustNet), Horizon 2020, 2020-2025.

References

[1] C. Avin, M. Bienkowski, A. Loukas, M. Pacut, and S. Schmid, *Dynamic balanced graph partitioning*, in SIAM SIAM Journal on Discrete Mathematics, vol. 34, 2019, pp. 1791–1812, https://doi.org/10.1137/17M1158513.

[2] C. Avin, M. Bienkowski, A. Loukas, M. Pacut, and S. Schmid, *Dynamic balanced graph partitioning*, SIAM Journal on Discrete Mathematics, 34 (2020), pp. 1791–1812, https://doi.org/10.1137/17M1158513.

[3] C. Avin, L. Cohen, M. Parham, and S. Schmid, *Competitive clustering of stochastic communication patterns on a ring*, in Journal of Computing, vol. 101, 2018, pp. 1369–1390, https://doi.org/10.1007/S00607-018-0666-X.

[4] C. Avin, M. Ghobadi, C. Griner, and S. Schmid, *On the complexity of traffic traces and implications*, Proceedings of the ACM on Measurement and Analysis of Computing Systems, 4 (2020), pp. 20:1–20:29, https://doi.org/10.1145/3379486.

[5] C. Avin, A. Loukas, M. Pacut, and S. Schmid, *Online balanced repartitioning*, in Proceedings of the 30th International Symposium on Distributed Computing (DISC 2016), vol. 9888 of Lecture Notes in Computer Science, Springer, 2016, pp. 243–256, https://doi.org/10.1007/978-3-662-53426-7_18.

[6] C. Bazgan, A. Nichterlein, and S. V. Alferez, *Destroying densest subgraphs is hard*, Journal of Computer and System Sciences, 151 (2025), p. 103635, https://doi.org/10.1016/J.JCSS.2025.103635.

[7] M. Bienkowski and S. Schmid, *A subquadratic bound for online bisection*, in Proceedings of the 41st International Symposium on Theoretical Aspects of Computer Science ((STACS) 2024), vol. 289 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:18, https://doi.org/10.4230/LIPICS.STACS.2024.14.

[8] T. Bläsius, P. Fischbeck, L. Gottesbüren, M. Hamann, T. Heuer, J. Spinner, C. Weyand, and M. Wilhelm, *PACE solver description: Kapoce: A heuristic cluster editing algorithm*, in 16th International Symposium on Parameterized and Exact Computation (IPEC 2021),

vol. 214 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 31:1–31:4, https://doi.org/10.4230/LIPICS.IPEC.2021.31.

[9] V. Cohen-Addad, D. R. Lolck, M. Pilipczuk, M. Thorup, S. Yan, and H. Zhang, *Combinatorial correlation clustering*, in Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC 2024), ACM, 2024, pp. 1617–1628, https://doi.org/10.1145/3618260.3649712.

[10] J. A. Enright and K. Meeks, *Deleting edges to restrict the size of an epidemic: A new application for treewidth*, Algorithmica, 80 (2018), pp. 1857–1889, https://doi.org/10.1007/s00453-017-0311-7.

[11] B. Feldkord, M. Feldotto, A. Gupta, G. Guruganesh, A. Kumar, S. Riechers, and D. Wajc, *Fully-dynamic bin packing with little repacking*, in Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), vol. 107 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 51:1–51:24, https://doi.org/10.4230/LIPICS.ICALP.2018.51.

[12] A. E. Feldmann and L. Foschini, *Balanced partitions of trees and applications*, Algorithmica, 71 (2015), pp. 354–376, https://doi.org/10.1007/s00453-013-9802-3.

[13] T. Forner, H. Raecke, and S. Schmid, *Online balanced repartitioning of dynamic communication patterns in polynomial time*, in Proceedings of 2nd Symposium on Algorithmic Principles of Computer Systems (APOCS 2020), SIAM, 2021, pp. 40–54, https://doi.org/10.1137/1.9781611976489.4.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[15] A. V. Goldberg, *Finding a maximum density subgraph*, tech. report, University of California at Berkeley, USA, 1984.

[16] M. Henzinger, S. Neumann, H. Raecke, and S. Schmid, *Tight bounds for online graph partitioning*, in Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA 2021), SIAM, 2021, pp. 2799–2818, https://doi.org/10.1137/1.9781611976465.166.

[17] M. Henzinger, S. Neumann, and S. Schmid, *Efficient distributed workload (re-)embedding*, 3 (2019), pp. 13:1–13:38, https://doi.org/10.1145/3322205.3311084.

[18] D. S. Hochbaum and D. B. Shmoys, *Using dual approximation algorithms for scheduling problems theoretical and practical results*, Journal of the ACM (JACM), 34 (1987), pp. 144–162.

[19] L. Kellerhals, T. Koana, A. Nichterlein, and P. Zschoche, *The PACE 2021 parameterized algorithms and computational experiments challenge: Cluster editing*, in 16th International Symposium on Parameterized and Exact Computation (IPEC 2021), vol. 214 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 26:1–26:18, https://doi.org/10.4230/LIPICS.IPEC.2021.26.

[20] Z. Lab, *Zhejiang lab llm trace program*. https://github.com/zhejianglab/Zhejiang-Lab-Cluster-Traces, 2024. Accessed: 2025-07-15.

[21] M. Pacut, M. Parham, and S. Schmid, *Optimal online balanced graph partitioning*, in Proceedings of the 40th IEEE Conference on Computer Communications (INFOCOM 2021), IEEE, 2021, pp. 1–9, https://doi.org/10.1109/INFOCOM42981.2021.9488824.

[22] J. Picard and M. Queyranne, *A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory*, Networks, 12 (1982), pp. 141–159, https://doi.org/10.1002/NET.3230120206.

[23] H. Räcke, S. Schmid, and R. Zabrodin, *Approximate dynamic balanced graph partitioning*, in Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), 2022, pp. 401–409, https://doi.org/10.1145/3490148.3538563.

[24] H. Räcke, S. Schmid, and R. Zabrodin, *Polylog-competitive algorithms for dynamic balanced graph partitioning for ring demands*, in Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2023), ACM, 2023, pp. 403–413, https://doi.org/10.1145/3558481.3591097, https://doi.org/10.1145/3558481.3591097.

[25] H. Räcke, S. Schmid, and R. Zabrodin, *Tight bounds for online balanced partitioning in the generalized learning model*, in Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA

2025), ACM, 2025, pp. 240–254, https://doi.org/10.1145/3694906.3743327, https://doi.org/10.1145/3694906.3743327.

[26] R. Rajaraman and O. Wasim, *Improved bounds for online balanced graph re-partitioning*, in Proceedings of the 30th Annual European Symposium on Algorithms (ESA 2022), vol. 244 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 83:1–83:15, https://doi.org/10.4230/LIPIcs.ESA.2022.83.

[27] P. Sanders, N. Sivadasan, and M. Skutella, *Online scheduling with bounded migration*, Math. Oper. Res., 34 (2009), pp. 481–498.

[28] L. G. Valiant, *Universality considerations in VLSI circuits*, IEEE Transactions on Computers, 30 (1981), pp. 135–140, https://doi.org/10.1109/TC.1981.6312176.

[29] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch, {*TopoOpt*}*: Co-optimizing network topology and parallelization strategy for distributed training jobs*, in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), USENIX Association, 2023, pp. 739–767, https://www.usenix.org/conference/nsdi23/presentation/wang-weiyang.